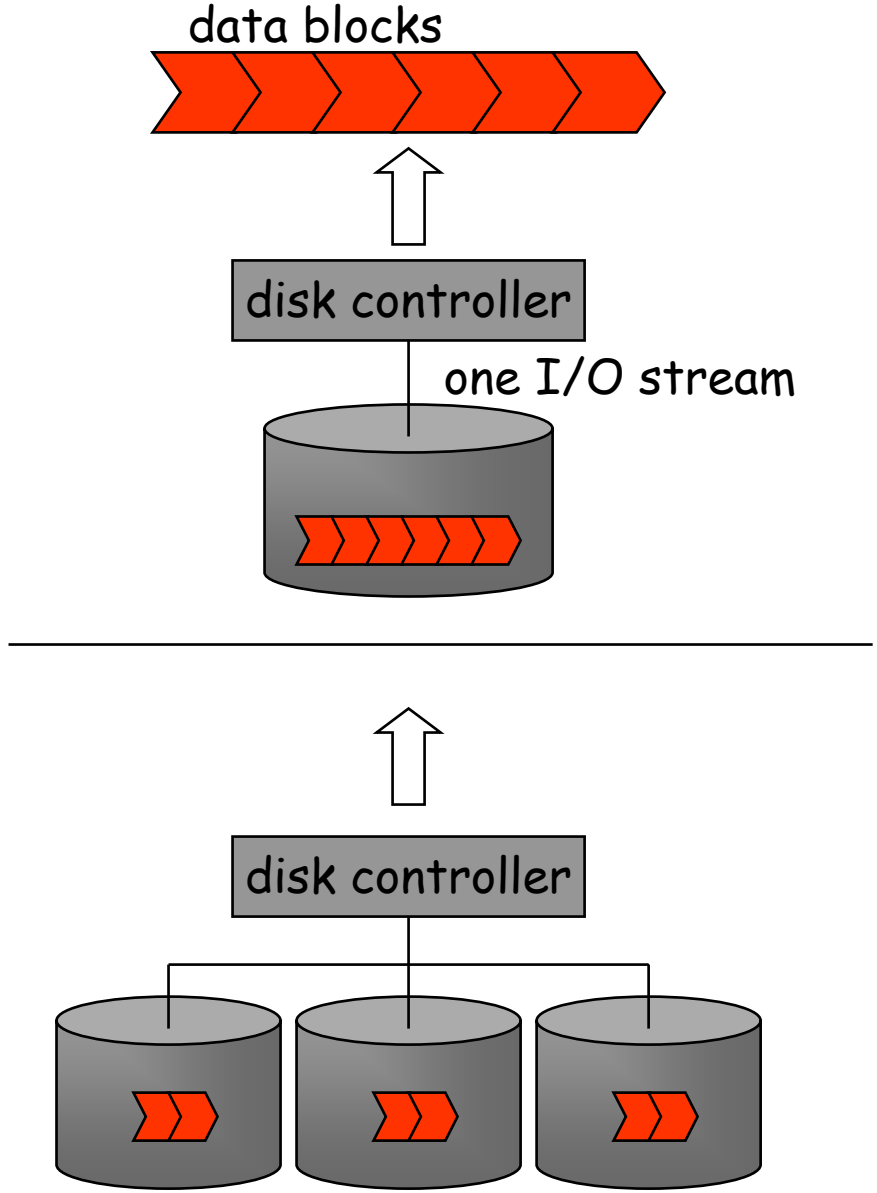
Data Striping

- The idea behind data striping is to distribute data among several disks so that it can be accessed in parallel
- Data striping takes place at a low system level (it is not user driven) and should be distinguished from data partitions in databases (which are user or application driven)
- Reasons for striping:
  - increase disk bandwidth by concurrently retrieving the data from several disks
  - decrease seek time (all disks do the seek in parallel)
  - handle several disk request in parallel
- Data striping is implemented in disk arrays or RAID systems



Fine vs coarse grain striping

FINE GRAIN

- Fine grained disk arrays use small data blocks so that all requests are serviced using all the disks at the same time
- The idea is to maximize the disk bandwidth (data transfer)
- The penalty for positioning the disk heads for every request is sequential, i.e., it must be paid for every request since requests are dealt with sequentially and all disks are used for every request
- Only one logical I/O request can be serviced at a time

COARSE GRAIN

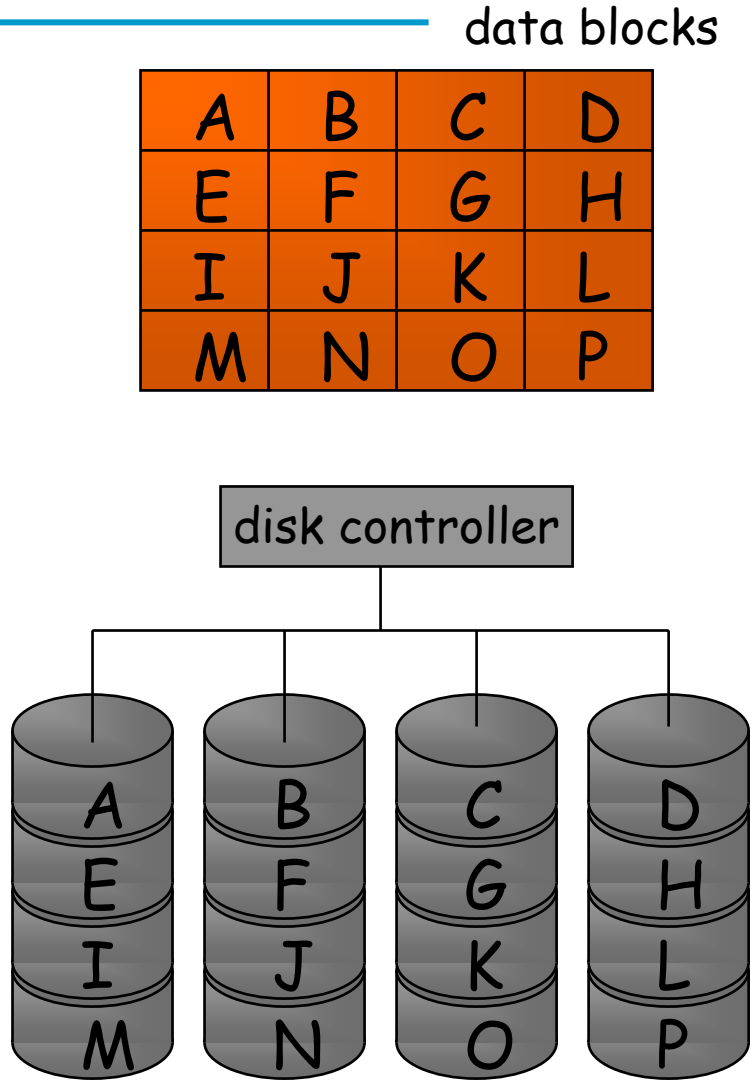
- Coarse grain disk arrays use large data blocks so that:
  - small request can be serviced in parallel since they will access only a few disks
  - large request can still benefit from high transfer rates by using many disks
- For small requests, the seek penalty is not sequential since several disks are used at the same time

Fault tolerance

- Disk arrays have the problem of using many independent disks:
  - probability of having a failure is the probability of any of the disks failing
  - if probability of a disk failing is P, the probability of a failure in a disk array with N disks is NxP
- Failures in disk arrays are dealt with by using redundancy and/or mirroring:
  - parity information is used to both detect and correct disk errors
  - mirroring is based on replication of data blocks
- The striping and parity depends on the block size:
  - bit-interleaved: each block is one bit, e.g., a byte can be stored in 8 disks. Parity is then on a per byte basis
  - block interleaved: each block contain several bytes (up to kbytes). Parity is on a per block basis
- The combination of parity and striping unit gives raise to the different RAID levels

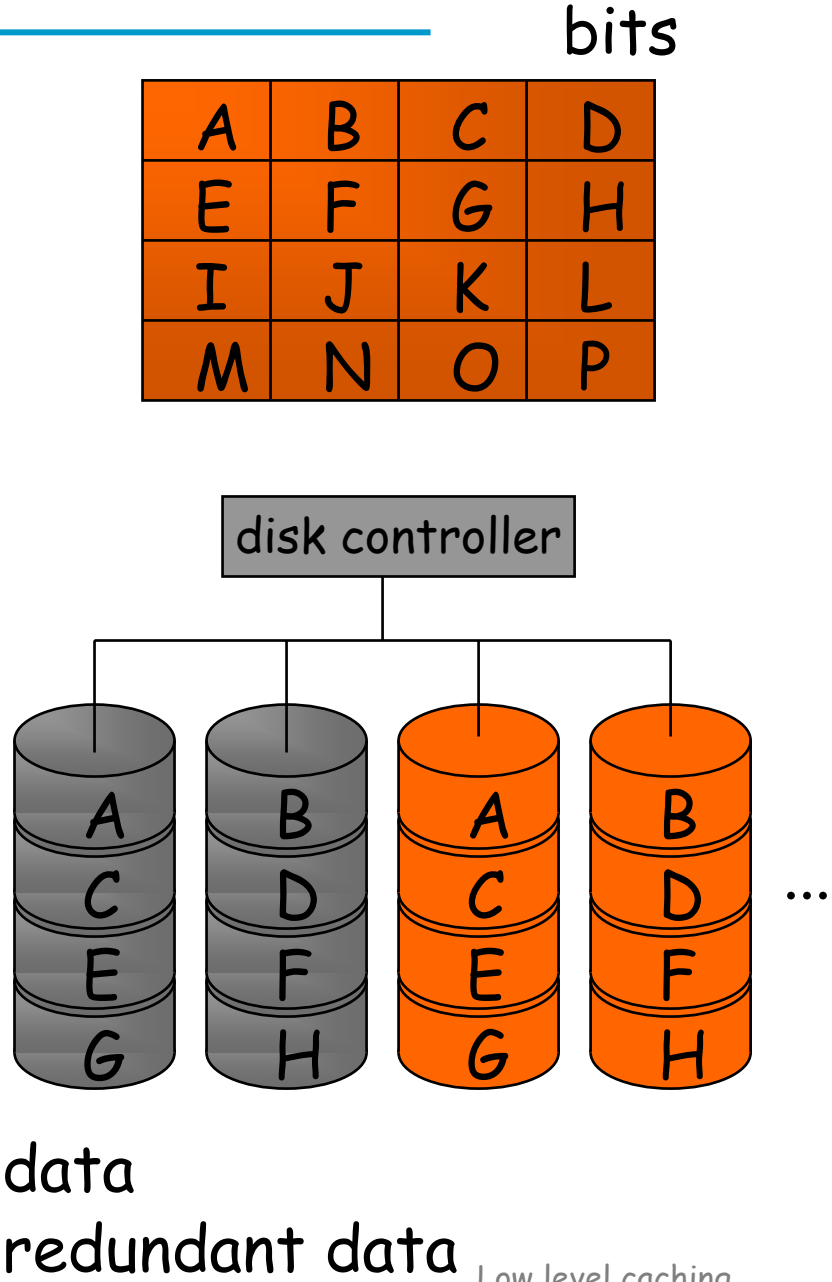
RAID level 0

- A RAID level 0 strips the data across the disks but without adding any redundancy
- The data is divided into blocks (of arbitrary size) and the blocks uniformly distributed across the disks
- I/O bandwidth is greatly improved (N times that of a single disk) for both reading and writing by using multiple disk channels in parallel
- A failure in one of the drives makes the entire RAID unavailable (no fault tolerance)
- Easily implemented in software
- No constraints on the number of disks



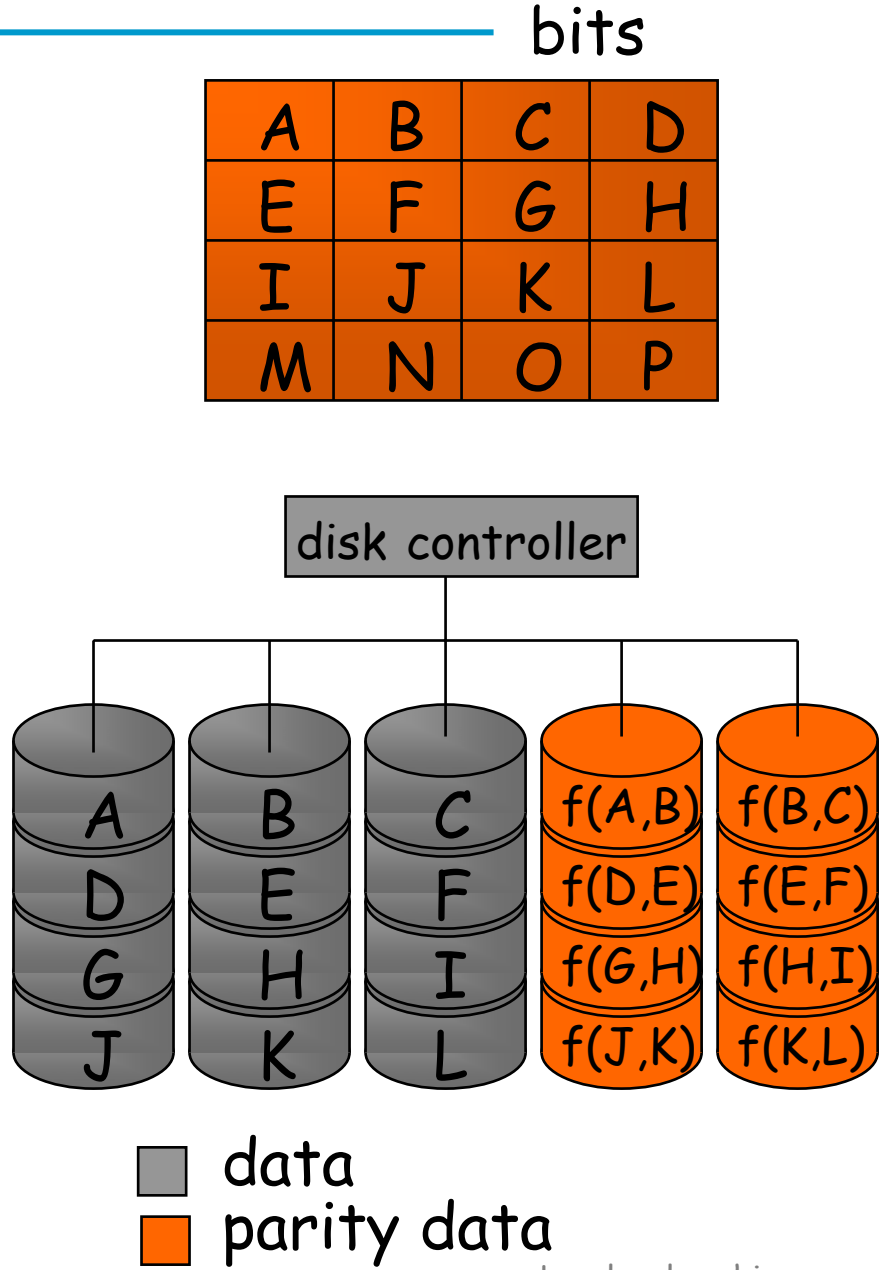
RAID level 1

- Bit interleaved
- Fault tolerance by mirroring (no parity)
- Read operations are performed on the copy that offers the smallest seek time
- Two read operations can be performed in parallel. Writes are sequential
- 50 % of the disk capacity is used for redundancy purposes. I/O bandwidth is only half of RAID level 0 (N/2)
- Recovery from failures is trivial
- Requires at least two disks (and twice as many as RAID level 0)
- It can handle multiple disk failures (as long as they are not on a mirrored pair)



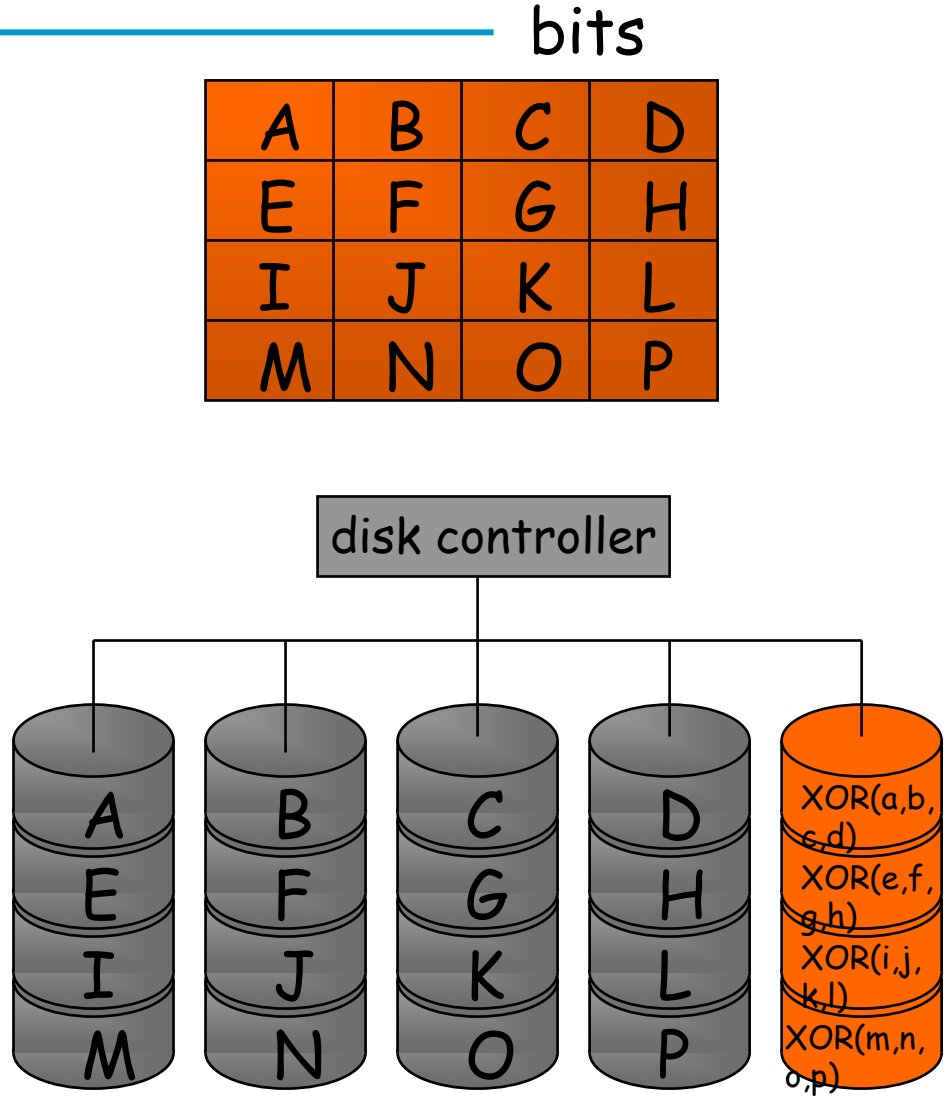
RAID level 2

- Bit interleaved
- Fault tolerance by mirroring based on Hamming codes
- Hamming codes implement parity for overlapping segments of data. They require less space than full mirroring but must be implemented in hardware
- Recovery is more complex (depends on the parity of several segments)
- I/O bandwidth is (N - log N), with log N being the number of disks needed for storing the parity
- It can handle multiple disk failures (depending on the failures)



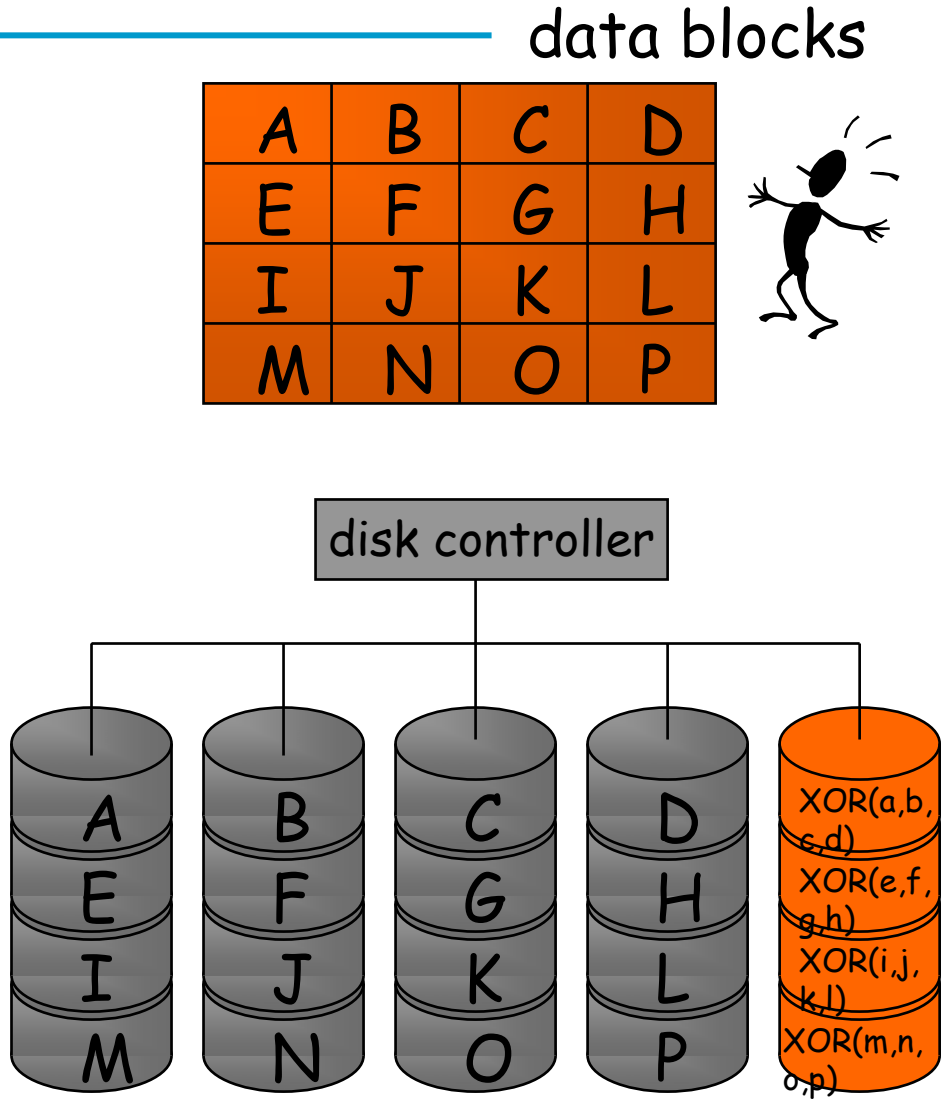
RAID level 3

- Bit interleaved
- There is a disk devoted to store the bit-wise parity of the other disks
- I/O bandwidth much better than levels 1 and 2 (N - 1).
- It can only handle one request at a time (no parallelism)
- Recovery is relatively simple (use parity to restore the data)
- Tolerates one disk failure
- This is fine grain stripping (adequate for applications that use large files, e.g., multimedia)



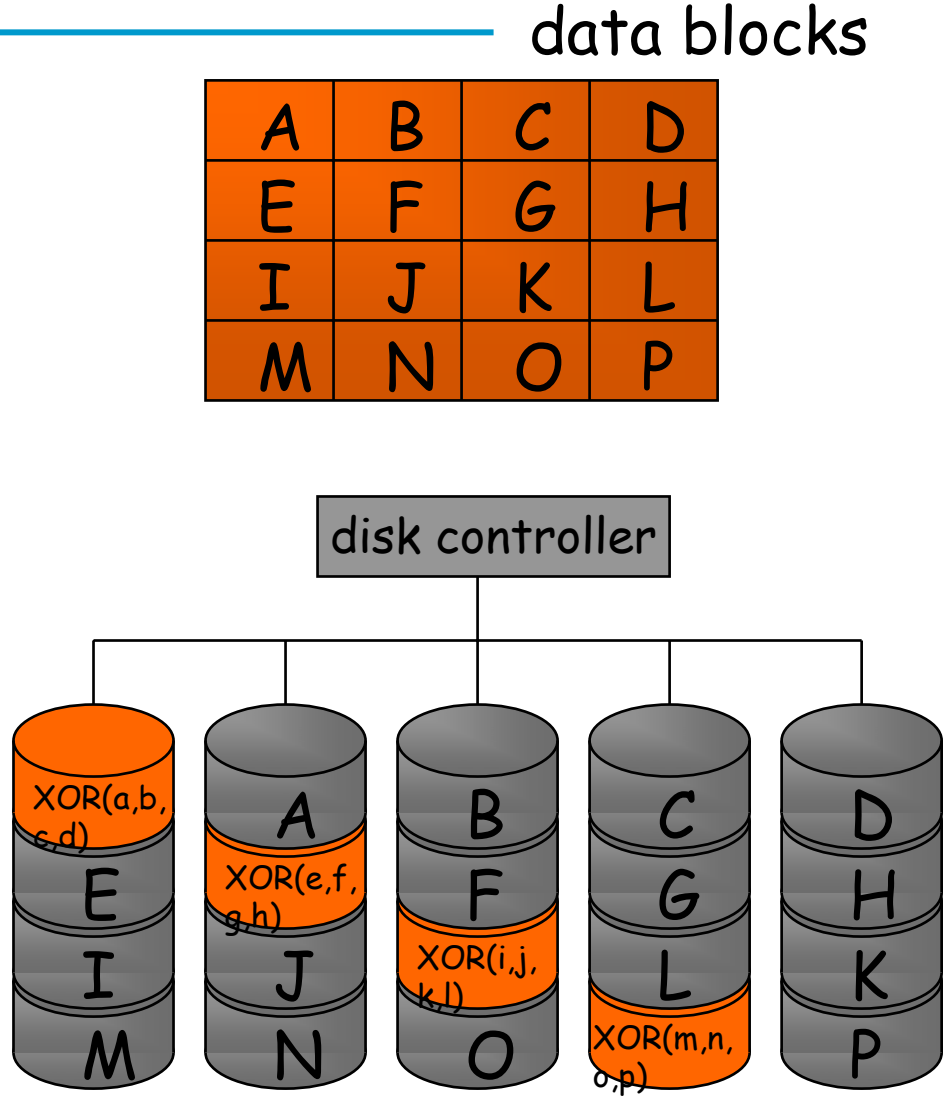
RAID level 4

- Block interleaved (blocks of arbitrary size, the size is called the striping unit)
- There is a disk devoted to store the block-wise parity of the other disks
- Write operations are sequential (all of them need to update the parity disk)
- Read operations can be done in parallel when on different blocks
- Parity disk is not used in read operations (limiting bandwidth)
- Tolerates one disk failure
- This is coarse grain stripping (adequate for standard databases with few update operations)

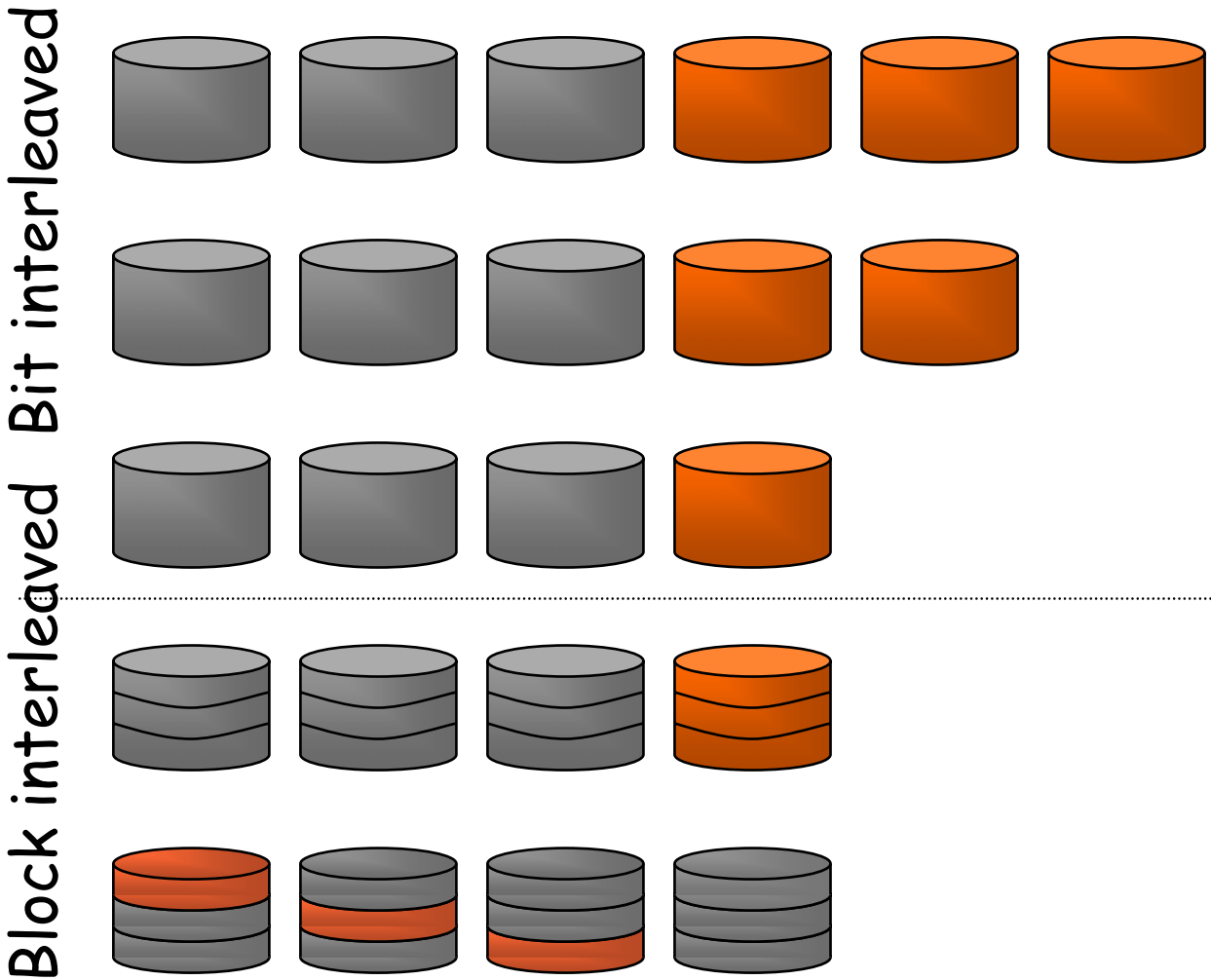


RAID level 5

- Block interleaved (blocks of arbitrary size, the size is called the striping unit)
- The block-wise parity is uniformly distributed across all disks
- Write operations can be done in parallel
- Read operations can be done in parallel when on different blocks
- Tolerates one disk failure, recovery is somewhat complex
- Overall good performance
- Small writes can be quite inefficient (because they require to read other blocks to complete the parity)
- Most popular approach (also in software)



Comparison of RAID levels



RAID level 1
(mirrored)

RAID level 2
(Hamming codes)

RAID level 3
(parity disk)

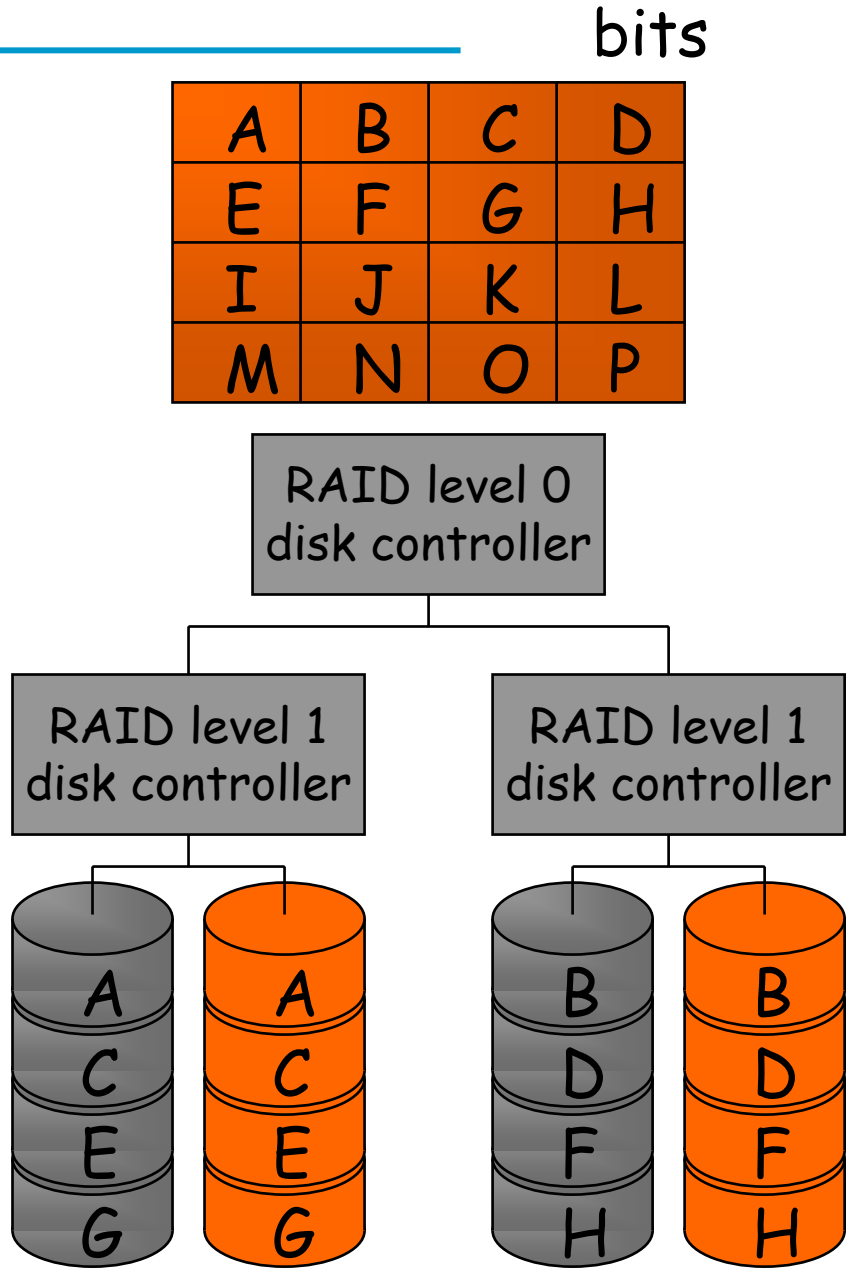
RAID level 4
(parity disk)

RAID level 5
(rotated parity)

Compare for small and large write and read operations ...

RAID level 10

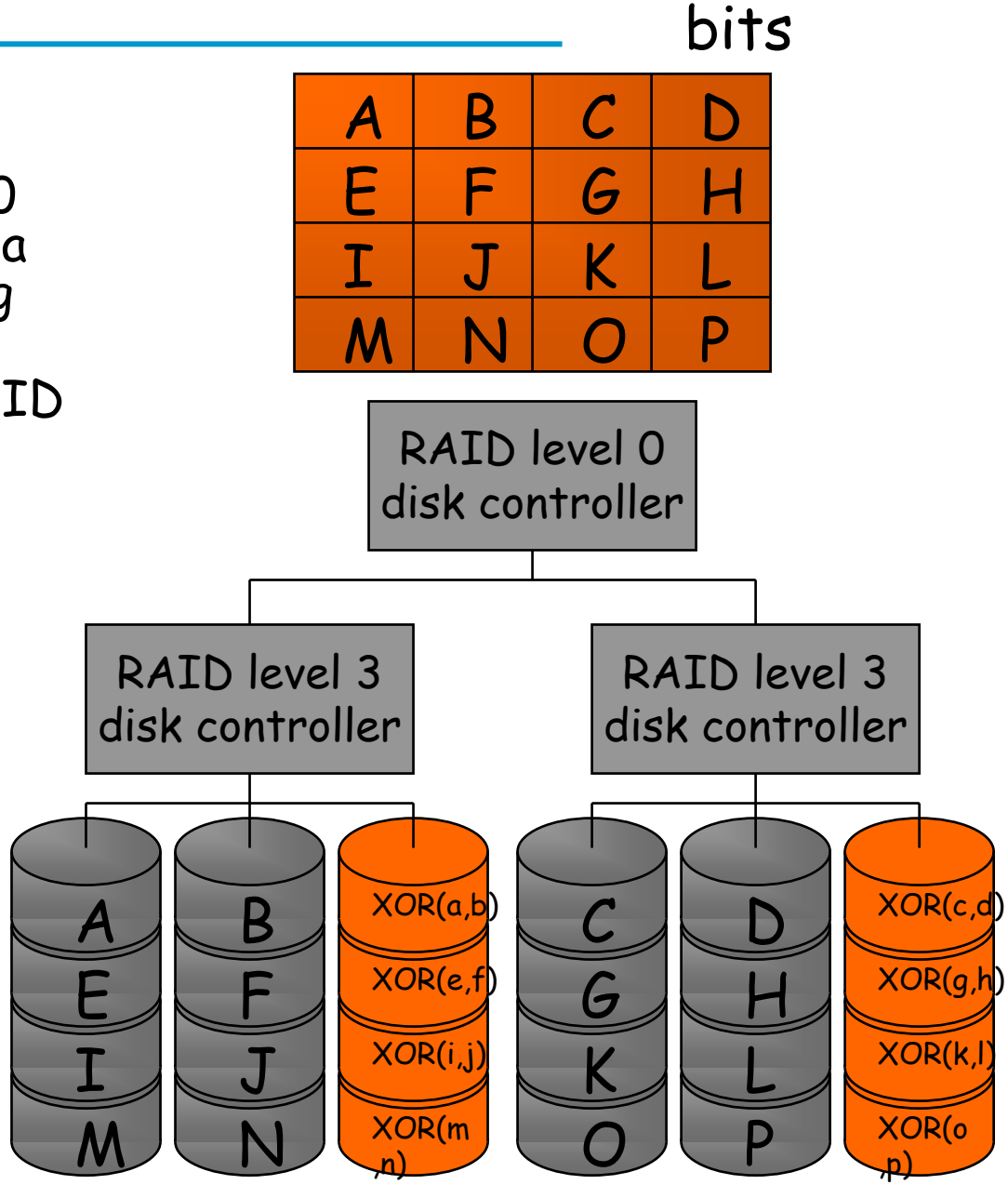
- RAID level 10 uses a RAID level 0 controller to strip the data. Each striping unit is then mirrored by a RAID level 1 controller
- Same fault tolerance as RAID level 1
- Requires at least 4 disks
- I/O bandwidth can be slightly better than level 1 because the level 1 controllers have less disks to manage



Low level caching

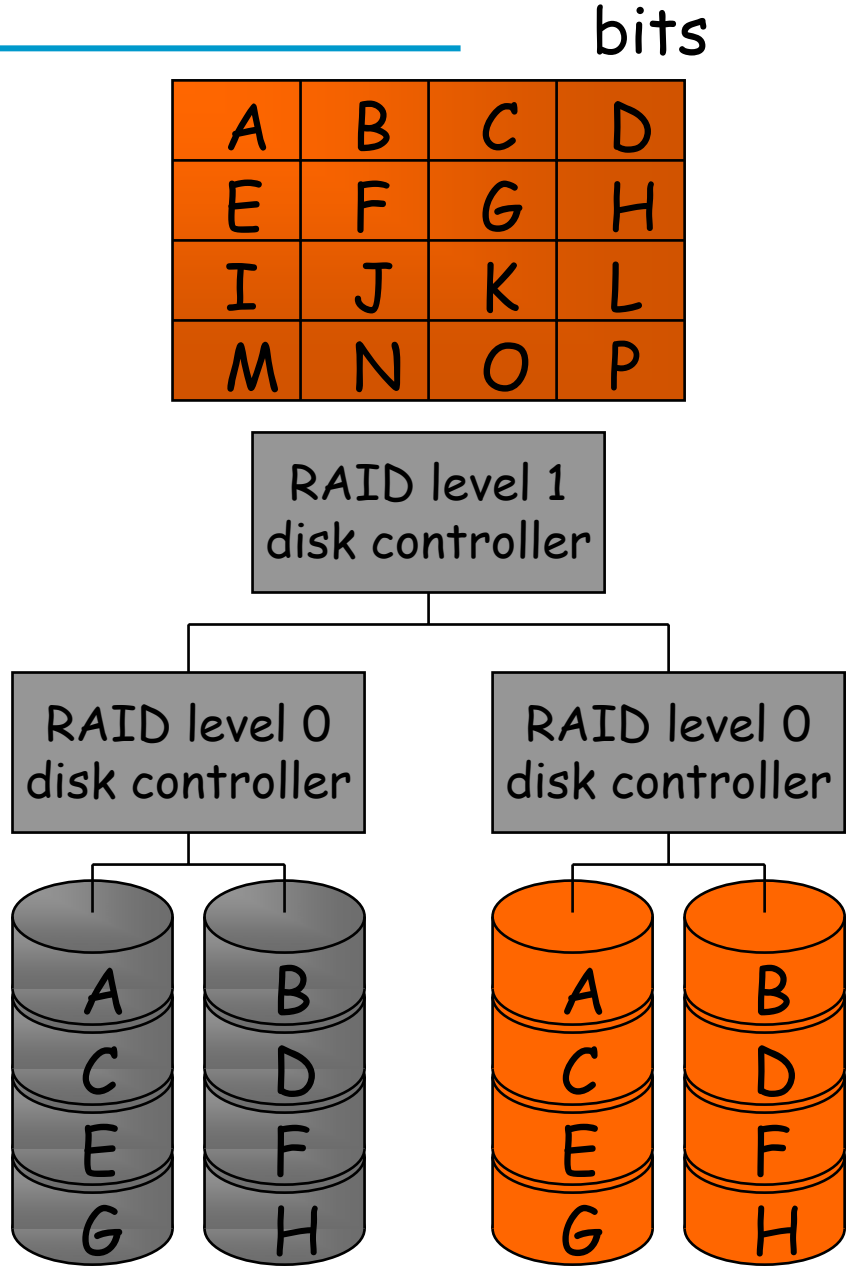
RAID level 53

- It has the wrong name (it should be 30)
- RAID level 53 uses a level 0 controller to stripe the data and then gives each striping unit to a level 3 controller
- Same fault tolerance as RAID level 3



RAID level 0 + 1

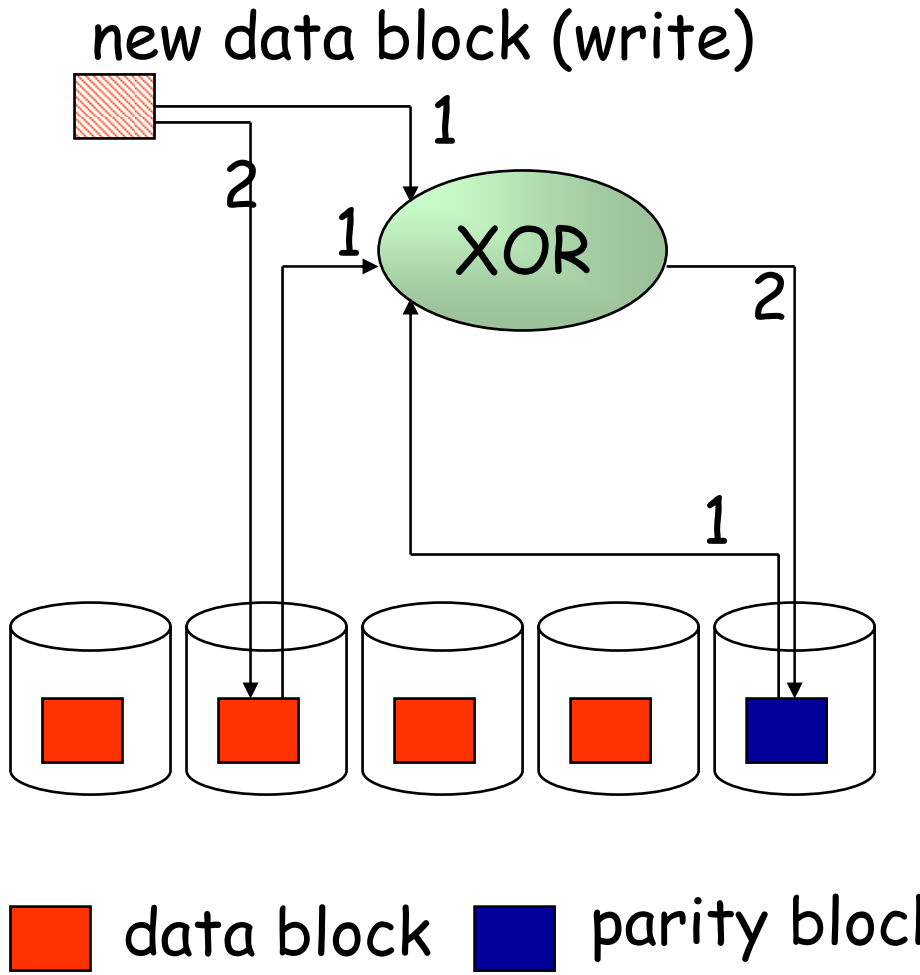
- RAID 0 + 1 uses a level 1 controller for mirroring the data and level 0 controllers for striping the mirrored disks
- Worse failure behavior that level 10
- It can execute reads in parallel (unlike level 10)



Low level caching

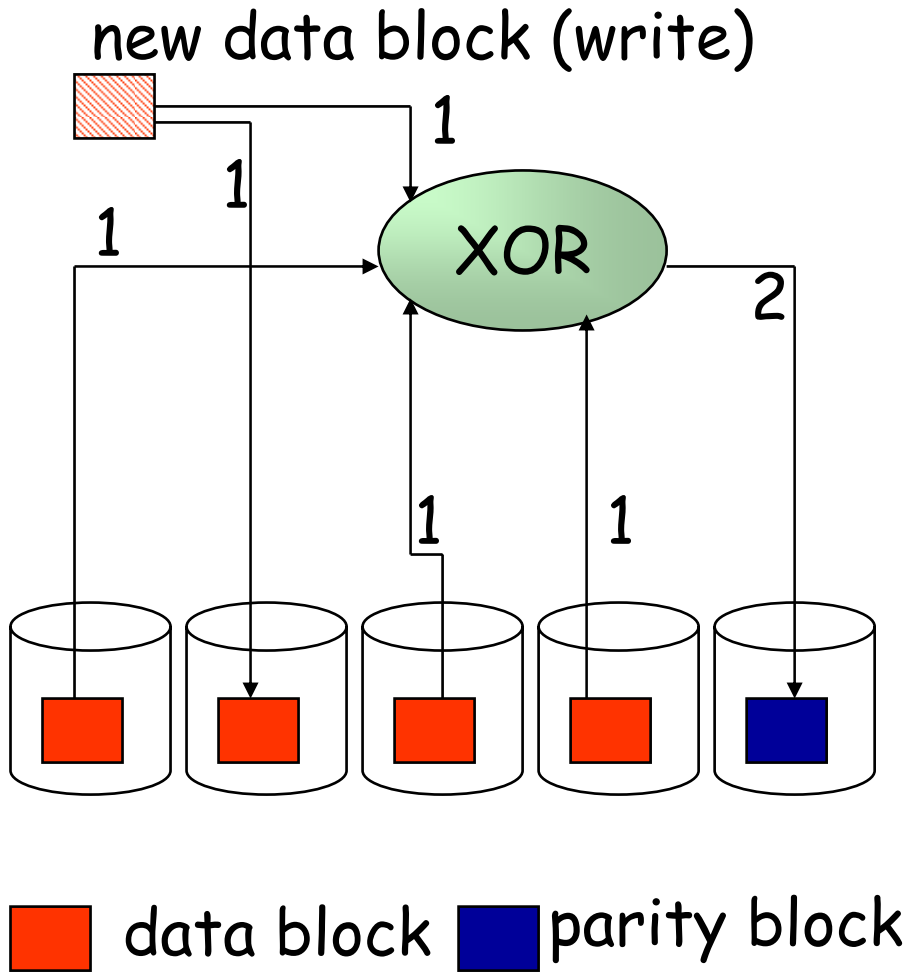
Small writes: read-write-modify

- In several RAID levels, small writes are a problem because they modify the parity but may not touch all the data fragments that define the parity
- Thus, small writes in these cases require to read all the data fragments that are needed for the parity even if they have nothing to do with the write operation itself
- The read-write-modify approach requires to read the data to modify before writing it. With the old data and the new data, the new parity can be computed without having to read all the fragments
- This allows to perform writes in parallel (depends on RAID level)

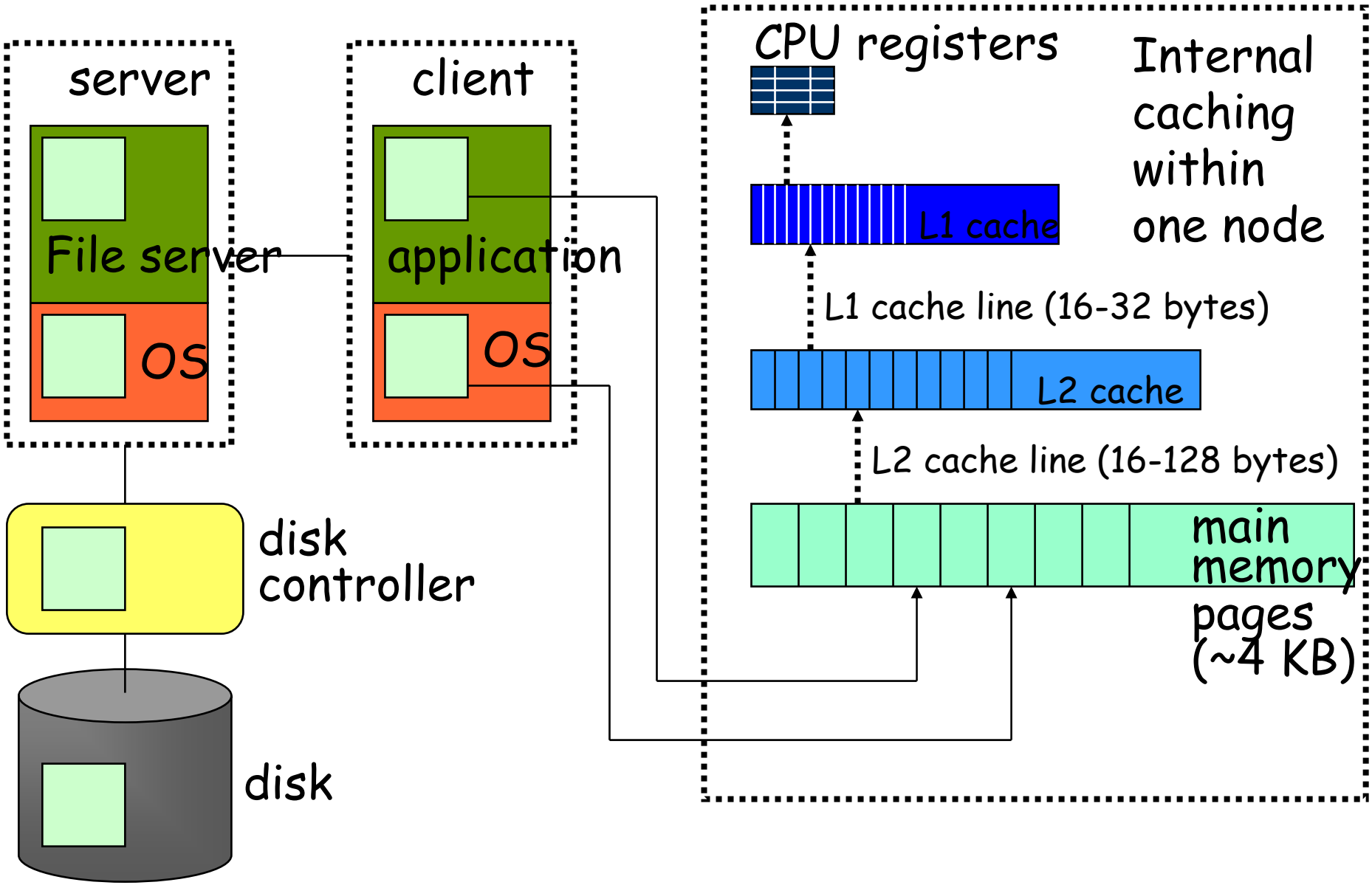


Small writes: regenerate-write

- An alternative is to read all data blocks needed for calculating the parity and then to regenerate the parity with the new data block
- With regenerate write, small writes use all the disks and can be performed only one at a time



More on caching

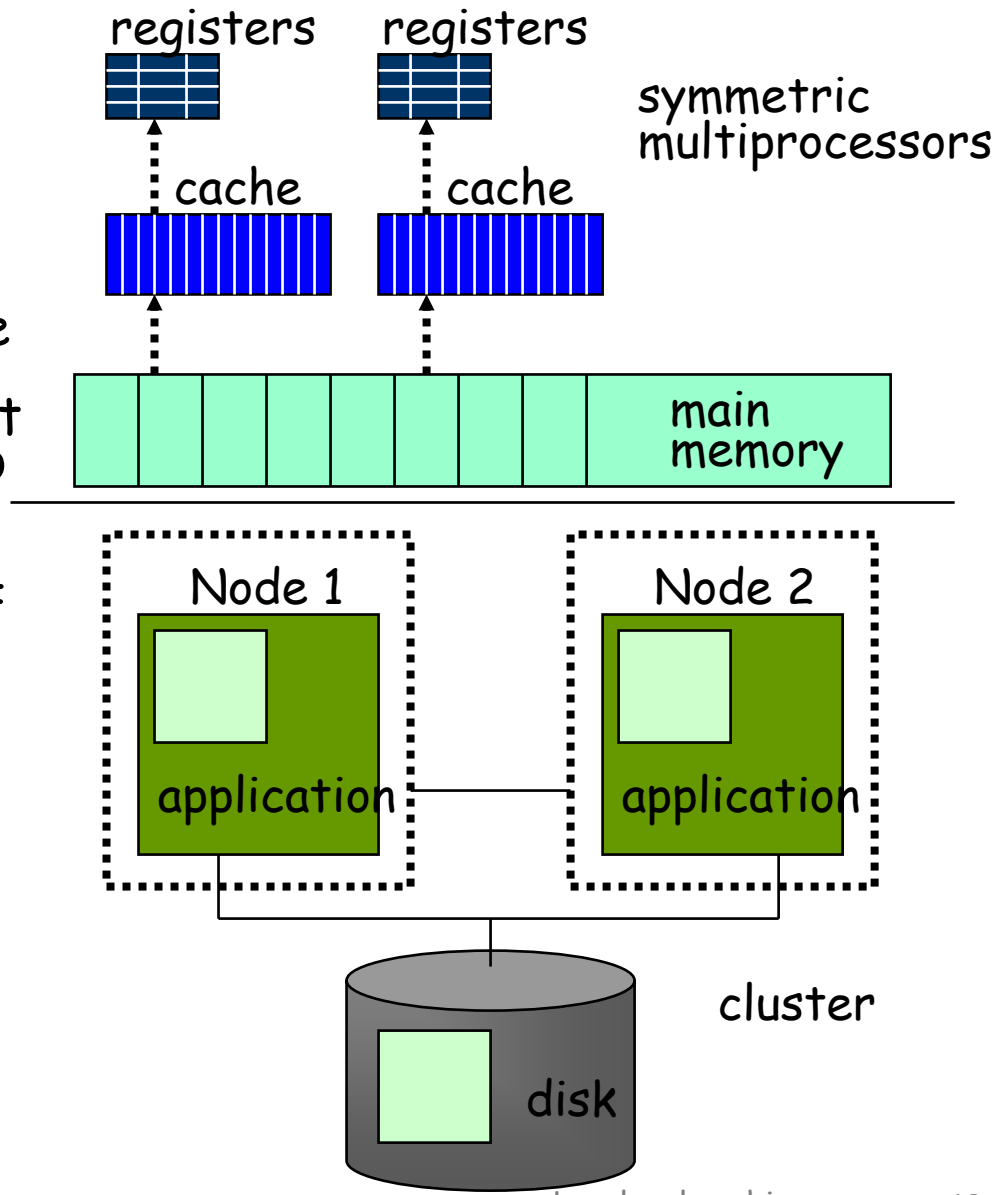


Multi-level caching

- Since disks are comparatively slow, disk controllers (particularly in RAID systems), provide a cache. With sufficient locality of access, this cache greatly speeds up access to the disk (and also facilitates writing, which can be done to the cache rather than to the disk directly)
- The same idea is applied at every level in the system. If the disks are accessed through a server, the server will cache data at both the operating system and the application (file server) level. The client will also cache data at the operating system and the application level (and might also cache data in its local disk)
- In a multilevel caching system, the closer to the hardware the cache is, the less effective the cache:
  - the reason is that locality is determined by the application, which is doing its own caching. Any further optimizations are done by the OS at the client machine, then by the file server, then the OS at the file server, etc.
  - this can be rephrased as follows: the further the cache from the source of locality, the less effective it will be
  - corollary: caching at the lower levels helps with multi-application access patterns not with application access patterns

Caches in parallel architectures

- When applications running on a multiprocessor machine are independent of each other, caching makes sure each application gets the data it needs
- When the application is a parallel application, chances are that each thread of execution may not be entirely independent of each other: they will need to access the same data
- Under such circumstances, caching results in replication of data blocks at different locations
- Like in any replication protocol, maintaining the coherency (consistency) of the copies is the main problem of cache based systems



Sessions and transactions

- Session (or snapshot) semantics
  - modifications to a data block are visible only on the node with the copy being modified. The other nodes do not see the changes
  - changes become visible once the data block is released (e.g., file is closed) but nodes must actively read the data block to observe the changes
  - nodes that do not read the data block again, will still see the old values
  - this is a last writer wins strategy
- Andrew File System uses these semantics

- Transaction semantics:
  - based on bracketing the operations with a BOT and EOT
  - any changes to a data block are not visible until EOT is executed
  - changes are propagated to all nodes upon transaction termination
- Typical of database systems

- Both approaches do not maintain consistency, it is the application developer who has to make sure things work properly

Tokens and leases

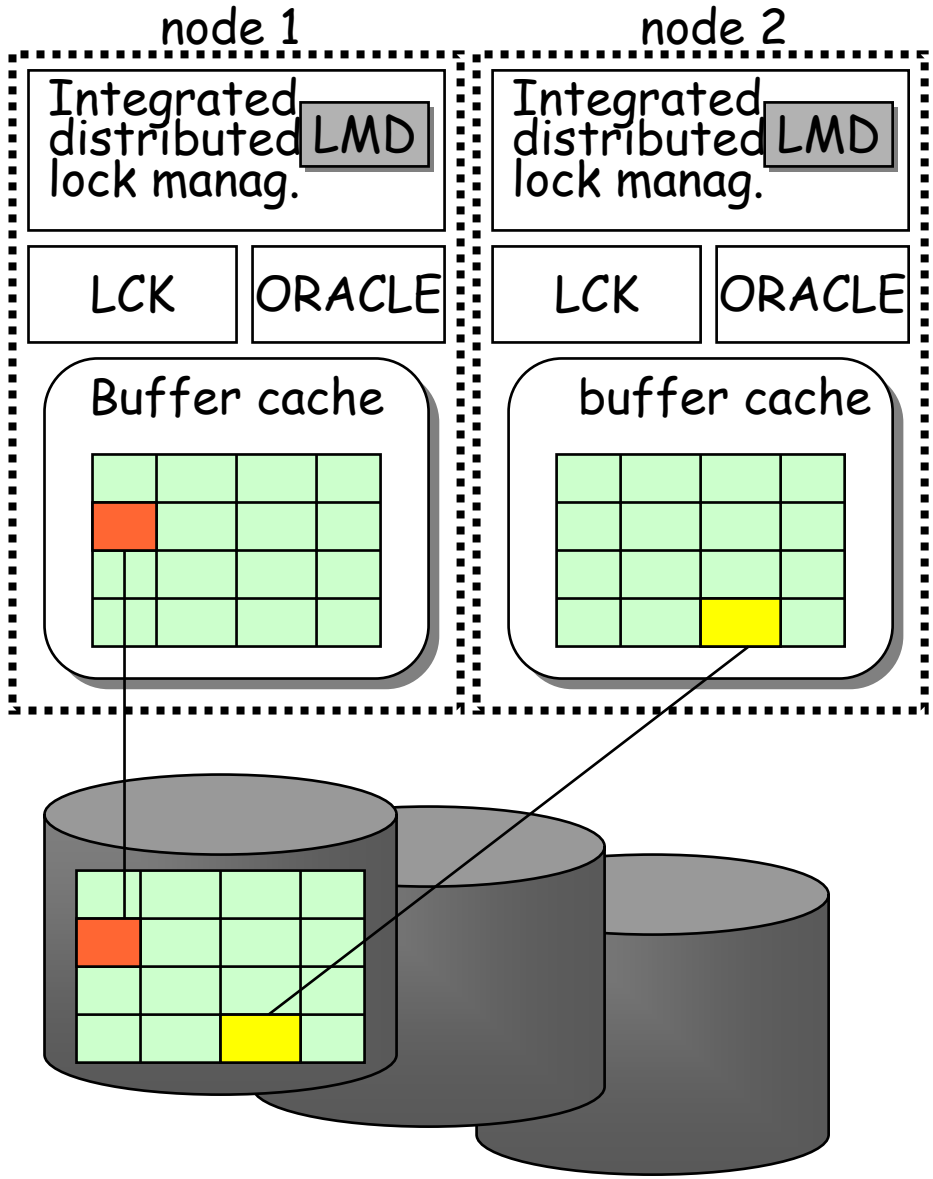
- Token based caching
  - in order to modify the local copy of a data block, a node needs the token
  - as long as a node has the token, it can freely modify the data block
  - if nobody is holding the token on that data block, any node can read the data block
  - when a node requests the token, copies in other caches are invalidated
  - problems arise if the token is lost (the node holding it fails or is disconnected from the rest of the system)
- Lease based caching
  - a lease is identical to a token but it has an expiration time
  - the lease (permission to write) is given only for a limited amount of time
  - after the time expires, the lease must be renewed
  - this prevents problems with failures
- Tokens and leases can be used at different granularities (files, blocks, or user defined)

Parallel Cache Management in Oracle

- Oracle has a shared disk architecture where nodes can locally cache data blocks from the disk
- The management of cached blocks is done through Parallel Cache Management locks (PCM locks):
  - PCM locks cover any type of data block on disk (data, index, undo, etc.)
  - PCM locks can be set at different block granularities (one or multiple blocks)
  - A PCM lock allows only one node to modify a block
  - After modification, any node wanting to read or modify the data block (i.e., wanting to set a PCM lock on it) must wait until the data block is written to disk
  - communication across nodes occurs in terms of pings: a ping occurs every time a block must be written to disk before another node can read it
  - locking and pinging are not related!! (if the data is properly partitioned, few pings will occur)
- How to deal with PCM locks is critical to obtain good performance

How PCM locks work

- When a node wants a block (module ORACLE determines which blocks to access), it requests a PCM lock on the block (through the module LCK)
- The integrated distributed lock manager creates or allocates the lock as needed
- If LCK needs to lock a block that has been locked by another node in exclusive mode, it does a ping on the remote node (the LMD module within the IDLM will contact the remote LMD module of the remote IDLM)
- Once the block is written to disk, it can be read from the disk (module ORACLE) and a lock set (module LCK)



Two types of PCM lock

Releasable

- releasable locks are locks that are dynamically allocated as needed and released when the block is no longer used
- to obtain a releasable lock, the lock must be first created, then obtained (more overhead)
- the advantage is that locks are not kept on data blocks if nobody is using them and nodes can start much faster
- Releasable locks can be hashed for faster performance

Fixed

- fixed locks are allocated at the start in the form of a hash table (the blocks are hashed to the locks). Upon creation, they are set to a null value
- fixed locks are kept until some other node makes a ping when they will be released and set to a null value (but the entry in the hash table remains)
- fixed locks are de-allocated only at shutdown

PCM locks and files

- The number of blocks covered by a PCM lock is determined by the user:
  - saying how many PCM locks correspond to that file (block-lock distribution done automatically)
  - by default: releasable locks with one PCM lock per block
  - several files can share PCM locks (be mapped to the same hash table of PCM locks)
  - A PCM lock can cover blocks in different files
- Locks do not directly map to blocks !!! (because blocks are hashed, they can be mapped anywhere in the hash table)

Assume A and B are 2 files with 44 blocks. We assign 44 locks

GC_FILES_TO_LOCKS="A,B:44"

32	33	34	35
36	37	38	39
40	41	42	43
44	1	2	3
4	5	6	7
8	9	10	11

28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	1	2	3

X 2 blocks per lock

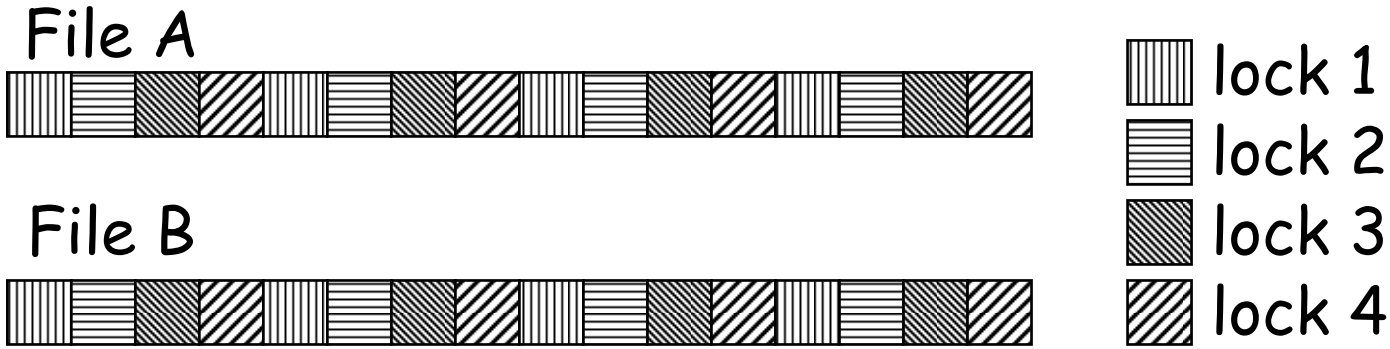
Y 1 block per lock

Locks 12-27 are not used

More examples of lock assignment

GC_FILES_TO_LOCKS = "1-2=4"

4 locks for the 2 files, blocks are hashed to the locks



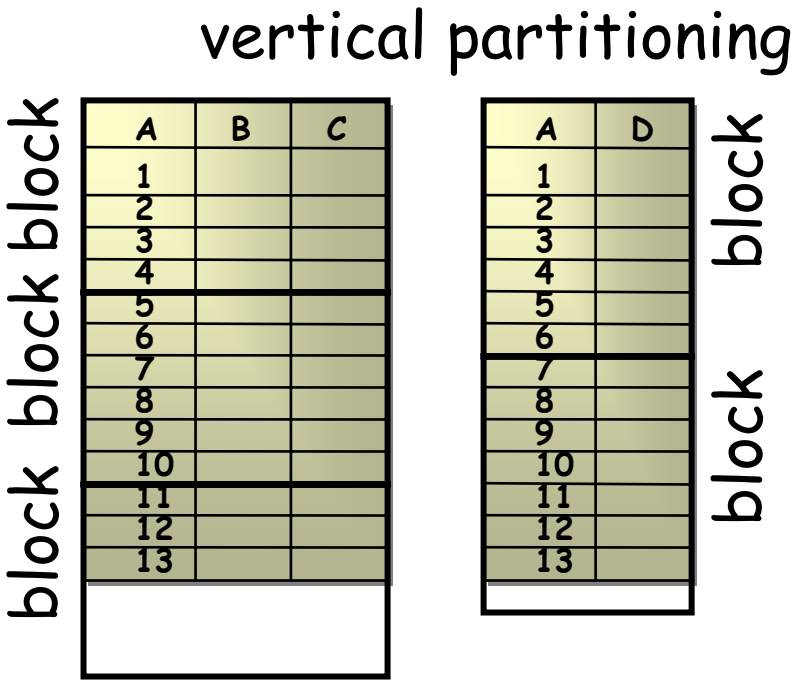
Useful if the files are used together as in

- if A is modified, B must also be modified
- A and B are always used to construct a bigger data table

However, no guarantee that the contents of the blocks under The same lock are actually correlated (see previous example). This means one might not be able to write A and B in parallel

Correlated tables and files

- Two or more files that are actually correlated is a common occurrence in parallel databases:
  - star schema: with a central table with all the important data and many auxiliary tables providing the necessary details
  - vertically partitioned tables: where the data is obviously correlated
- In these cases, it does not make much sense to access the auxiliary tables by themselves, they are only accessed (if at all) as a result of a search on the main table
- Associating the locks for both is an advantage:
  - less locks are needed

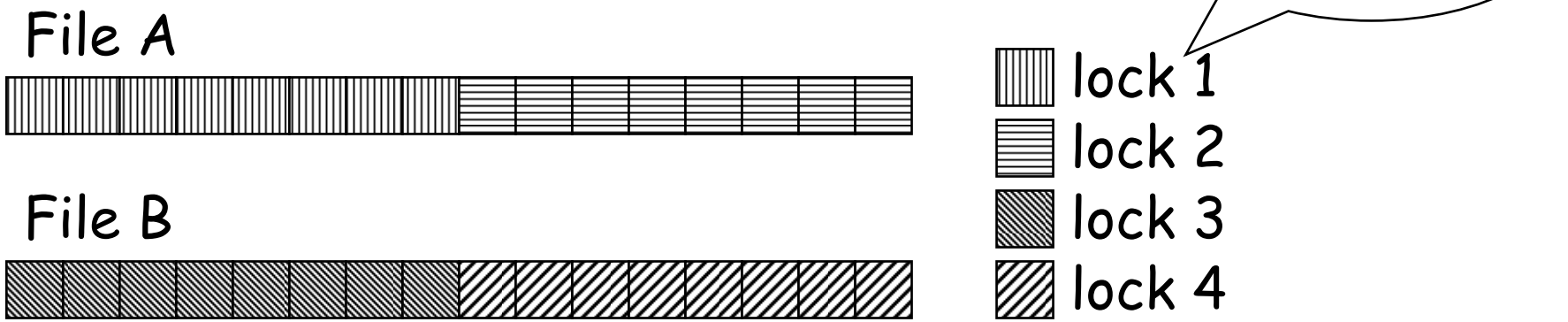


WARNING: the blocks are not aligned with the data. Same principle as in multilevel caching

More examples of lock assignment

GC_FILES_TO_LOCKS = "1-2=4!8"

4 locks for the 2 files, blocks under one lock must be contiguous



Useful for operations that will modify or scan long ranges:

- in principle, both files can be modified in parallel

Alignment is not guaranteed: if a file does not have a multiple of the size of continuous blocks, some locks will lock more blocks than others

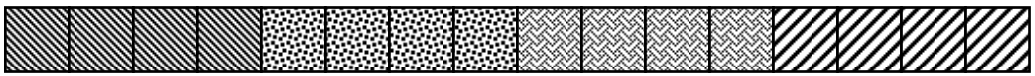
More examples of lock assignment

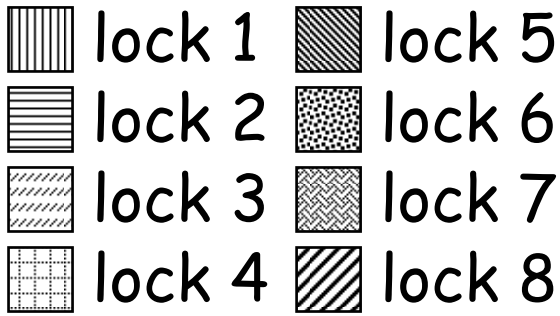
GC_FILES_TO_LOCKS = "1-2=4!4EACH"
4 locks for each of the 2 files, blocks under one lock must be contiguous

Assume A and B are 2 files with 16 blocks each

File A

File B





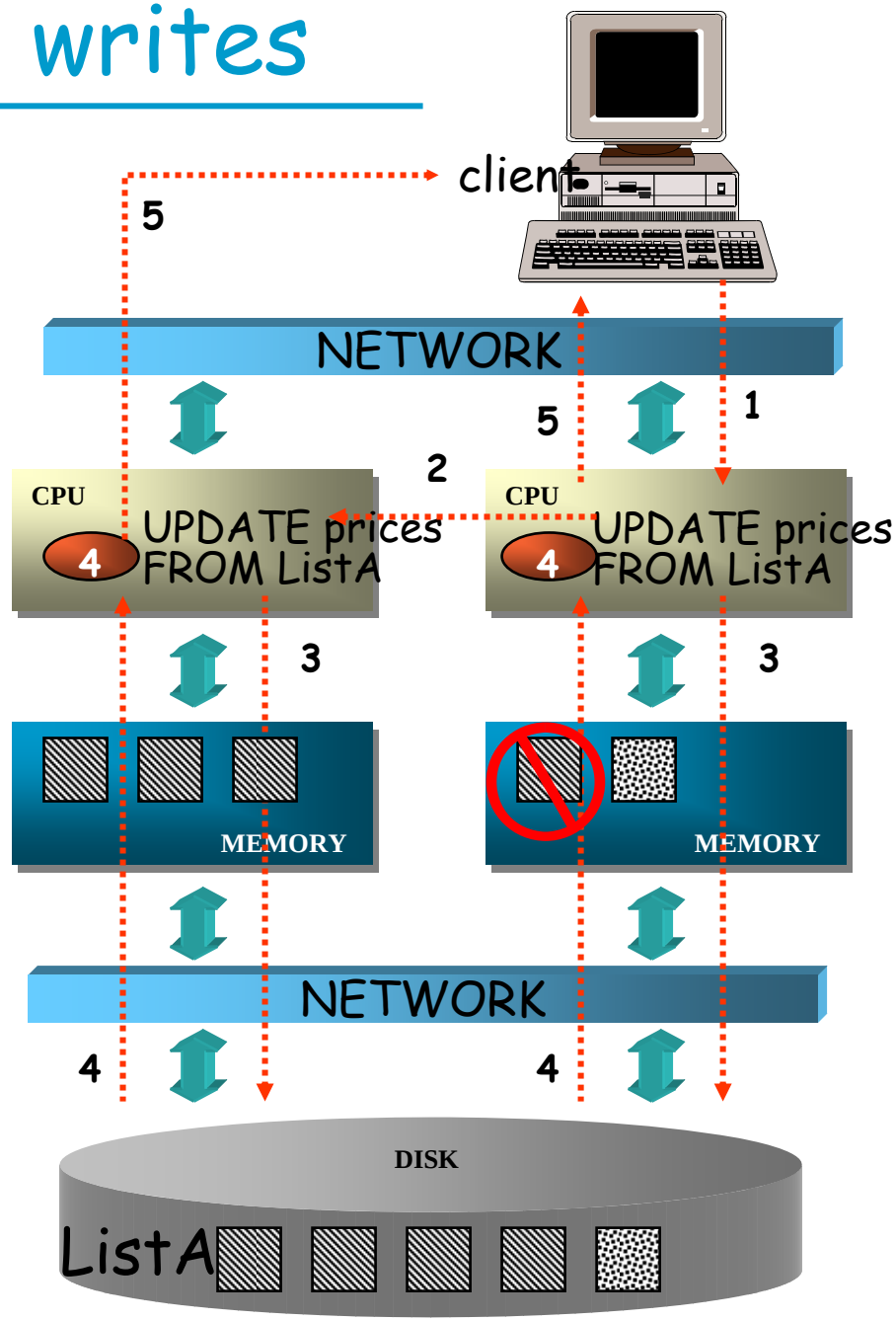
Useful for operations that will modify or scan medium size ranges:

- in principle, both files can be modified in parallel

Alignment is not guaranteed: if a file does not have a multiple of the size of continuous blocks, some locks will lock more blocks than others

The curse of parallel writes

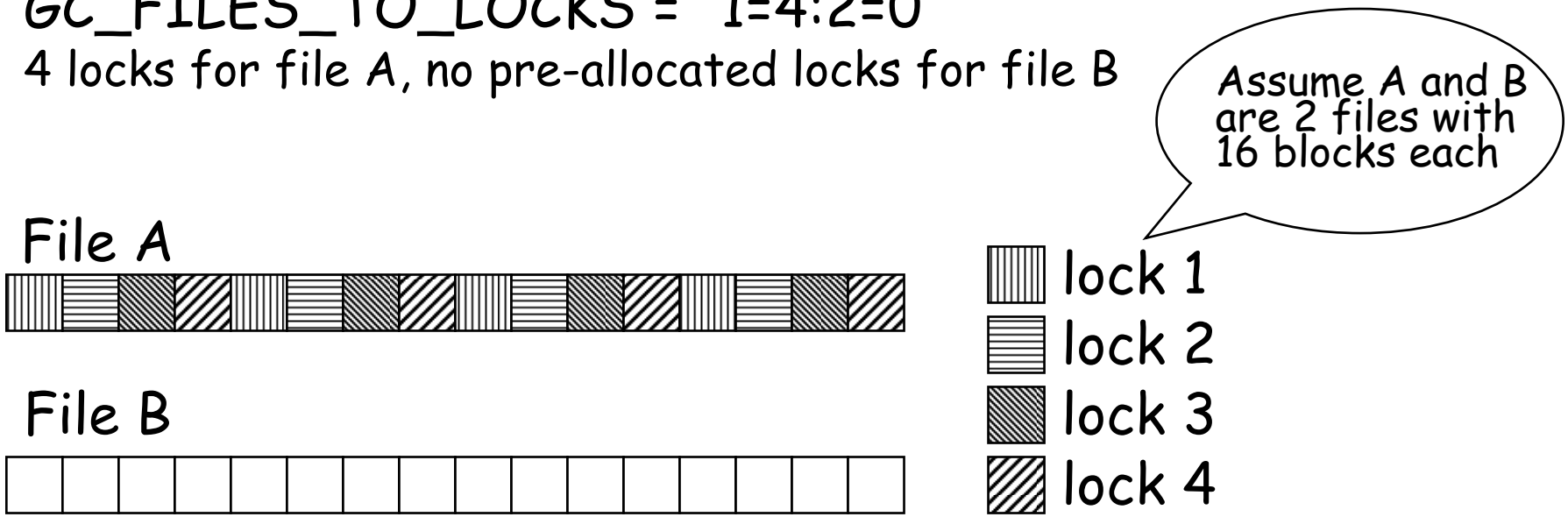
- Parallel writes pose a serious problem for disk caching independently of the locking granularity:
  - data and disk blocks are not aligned
  - there is no clear way to partition the load for function shipping and execution in parallel
- This demonstrates the same principle:
  - caching at the lower levels helps with multi-application access patterns not with application access patterns. In this case, the cache helps to execute different functions in parallel, it makes it very difficult to parallelize the updates of a single function
  - only solution is to physically partition the table



More examples of lock assignment

GC_FILES_TO_LOCKS = "1=4:2=0"

4 locks for file A, no pre-allocated locks for file B



In file A, the locks contain several blocks but the blocks are not consecutive. In file B, the locks are not pre-allocated. They are allocated on demand at the rate of one lock per block

Why releasable locks?

- Releasable locks are at the finest granularity level
- Maintaining a hash table at the block granularity level for a file (a table) is very expensive since it may require tens of thousand of entries; thus, fixed locks tend to be on multiple blocks
- When a PCM lock cover multiple blocks, it might introduce false conflicts between update operations (updates to actually different blocks but that covered by a single lock and, therefore, cannot be modified concurrently)
- With releasable locks:
  - false conflicts are minimized
  - no ping is necessary if nobody is holding the lock (useful for tables with a lot of update traffic)
  - there is a clear overhead per block access (lock must be created, set and the released)
- Releasable locks are there for those cases where fixed locks do not work well